Занятие 9

«Многопоточность»

Состав

- Понятие многопоточности
- Способы запуска потоков
- Способы синхронизации потоков
- Deadlock
- Модель памяти Java
- Пакет java.util.concurrent

Понятие многопоточности

Многопоточность- свойство платформы (например, операционной системы, виртуальной машины и т. д.) или приложения, состоящее в том, что процесс, порождённый в операционной системе, может состоять из нескольких потоков, выполняющихся «параллельно», то есть без предписанного порядка во времени.

Процесс — это совокупность кода и данных, разделяющих общее виртуальное адресное пространство.

Поток- наименьшая последовательность команд, которая может исполняться независимо друг от друга.

Один процесс может может включать от 1 до нескольких потоков.

Мотивация

- Эффективное использование ресурсов железа (в частности процессора)
- Упрощение кода программы за счет распараллеливание однотипных действий

Реализация

- Аппаратная (не рассматривается в курсе)
- Программная

Запуск потоков

1 Вариант- имплементация Runnable

```
public class MyRunnable implements Runnable {

    public void run(){
      System.out.println("MyRunnable running");
    }
  }

Thread thread = new Thread(new MyRunnable());
thread.start();

Runnable myRunnable = new Runnable(){

     public void run(){
       System.out.println("Runnable running");
     }
   }

   Thread thread = new Thread(myRunnable);
```

```
thread.start();
```

2 Вариант- наследование Thread

```
public class MyThread extends Thread {

  public void run(){
    System.out.println("MyThread running");
  }
 }

MyThread myThread = new MyThread();
 myTread.start();

 Thread thread = new Thread(){
  public void run(){
   System.out.println("Thread Running");
  }
 }

 thread.start();
```

3 Вариант- имплементация Callable

Callable подобен Runnable, но с возвратом значения. Интерфейс Callable является параметризованным типом, с единственным общедоступным методом call().

```
public interface Callable<V>;
{
        V call() throws Exception;
}
```

Параметр представляет собой тип возвращаемого значения. Например, Callableпредставляет асинхронное вычисление, которое в конечном итоге возвращает объект Integer.

Future хранит результат асинхронного вычисления. Вы можете запустить вычисление, предоставив кому-либо объект Future, и забыть о нем. Владелец объекта Future может получить результат, когда он будет готов.

Интерфейс Future имеет следующие методы.

```
public interface Future<V>
{
    V get() throws . . .;
    V get(long timeout, TimeUnit unit) throws . . .;
    boolean isCancelled();
    boolean isDone();
}
```

Вызов первого метода get() устанавливает блокировку до тех пор, пока не завершится вычисление. Второй метод генерирует исключение TimeoutException, если истекает таймаут до завершения вычислений. Если прерывается поток, выполняющий вычисление, оба метода генерируют исключение InterruptedException. Если вычисление уже завершено, get() немедленно возвращает управление.

Метод isDone() возвращает false, если вычисление продолжается, и true — если оно завершено.

Вы можете прервать вычисление,вызвав метод Cancel(). Если вычисление еще не стартовало, оно отменяется и уже не будет запущено. Если же вычисление уже идет, оно прерывается в случае равенства true параметра mayInterrupt.

Класс-оболочка FutureTask представляет собой удобный механизм для превращения Callable одновременно в Future и Runnable, реализуя оба интерфейса.

Например:

```
Callable<Integer> myComputation = . . .;
FutureTask<Integer> task = new FutureTask<Integer>(myComputation);
Thread t = ne Thread(task);  // это Runnable
t.start();
...
Integer result = task.get();  // это Future
```

Далее,. мы просто будем подсчитывать количество файлов, соответствующих критерию поиска. Таким образом, у нас будет долго работающая задача, которая в результате даст целочисленное значение — пример Callable.

```
class MatchCounter implements Callable<Integer>{
    public MatchCounter(File directory, String keyword) { . . . }
    public Integer call() { . . . }  // возвращает количество найденных файлов
}
```

Сконструируем объект из MatchCounter и используем его для запуска потока:

```
FutureTask<Integer>(counter);
Thread t = new Thread(task);
t.start();
```

И, наконец, напечатаем результат:

```
System.out.println(task.get() + " файлов найдено.");
```

Вызов get() устанавливает блокировку до тех пор, пока не будет готов результат.

Внутри метода call() мы используем тот же механизм рекурсивно. Для каждого подкаталога создадим новый MathCounter и запустим поток для него. Также мы спрячем объект FutureTask в ArrayList<future></future. И в конце сложим все результаты:

```
for (Future<Integer> result : results)
count += result.get();
```

Каждый вызов get() устанавливает блокировку до тех пор, пока не будет готов результат. Но, потоки работают параллельно, так что есть шанс, что результаты будут готовы почти одновременно.

Синхронизация

Каждый объект в java имеет свой монитор. Для достижения эффектов взаимного исключения и синхронизации потоков используют следующие операции:

- monitorenter: захват монитора. В один момент времени монитором может владеть лишь один поток. Если на момент попытки захвата монитор занят, поток, пытающийся его захватить, будет ждать до тех пор, пока он не освободится. При этом, потоков в очереди может быть несколько.
- monitorexit: освобождение монитора

- wait: перемещение текущего потока в так называемый wait set монитора и ожидание того, как произойдёт notify. Выход из метода wait может оказаться и ложным. После того, как поток, владеющий монитором, сделал wait, монитором может завладеть любой другой поток.
- notify(all): пробуждается один (или все) потоки, которые сейчас находятся в wait set монитора. Чтобы получить управление, пробуждённый поток должен успешно захватить монитор (monitorenter)

Определение:

Contention — ситуация, когда несколько сущностей одновременно пытаются владеть одним и тем же ресурсом, который предназначен для монопольного использования. От того, есть ли contention на владение монитором, зависит то, как производится его захват. Монитор может находиться в следующих состояниях:

- init: монитор только что создан, и пока никем не был захвачен
- biased: (умная оптимизация) Монитор «зарезервирован» под первый поток, который его захватил. В дальнейшем для захвата этому потоку не нужны дорогие операции, и захват происходит очень быстро. Когда захват пытается произвести другой поток, либо монитор пере-резервируется для него (rebias), либо монитор переходит в состояние thin (revoke bias). Также есть дополнительные оптимизации, которые действуют сразу на все экземпляры класса объекта, монитор которого пытаются захватить (bulk revoke/rebias)
- thin: монитор пытаются захватить несколько потоков, но contention нет (т.е. они захватывают его не одновременно, либо с очень маленьким нахлёстом). В таком случае захват выполняется с помощью сравнительно дешёвых CAS. Если возникает contention, то монитор переходит в состояние inflated
- fat/inflated: синхронизация производится на уровне операционной системы. Поток паркуется и спит до тех пор, пока не настанет его очередь захватить монитор. Даже если забыть про стоимость смены контекста, то когда поток получит управление, зависит ещё и от системного шедулера, и потому времени может пройти существенно больше, чем хотелось бы. При исчезновении contention монитор может вернуться в состояние thin

Заголовки объектов

Внутри виртуальной машины заголовки объектов в общем случае содержат два поля: mark word и указатель на класс объекта. В частных случаях туда может что-то добавляться: например, длина массива. Хранятся эти заголовки в так называемых oop-ах (Ordinary Object Pointer), и посмотреть на их структуру можно в файле hotspot/src/share/vm/oops/oop.hpp. Мы же более детально изучим то, что из себя представляет mark word, который описывают в файле markOop.hpp, находящемся в той же папке. (Не обращайте внимание на наследование от oopDesc: оно есть лишь по историческим причинам) По-хорошему, его стоит вдумчиво почитать, уделив внимание подробным комментариям, но для тех, кому это не очень интересно, ниже краткое описание того, что же в этом mark word содержится и в каких случаях. Можно ещё посмотреть вот эту презентацию начиная с 90-го слайда.

Содержимое mark words

Состояние	Тег	Содержимое			
unlocked, thin, unbiased	01	Identity hashcode	Age		0
locked, thin, unbiased	00	указатель на место нахождения mark word			
inflated	10	указатель на раздутый монитор			
biased	01	id потока-владельца	epoch	age	1
marked for GC	11				

Здесь мы видим несколько новых значений. Во-первых, identity hash code — тот хеш-код объекта, который возвращается при вызове System.identityHashCode. Во-вторых, age — сколько сборок мусора пережил объект. И ещё есть epoch, которая указывает число bulk revocation или bulk rebias для класса этого объекта. К тому, зачем это нужно, мы подойдём позже.

Вы заметили, что в случае biased не хватило места одновременно и для identity hash code и для threadID + epoch? А это так, и отсюда есть интересное следствие: в hotspot вызов System.identityHashCode приведёт к revoke bias объекта.

Далее, когда монитор занят, в mark word хранится указатель на то место, где хранится настоящий mark word. В стеке каждого потока есть несколько «секций», в которых хранятся разные вещи. Нас интересует та, где хранятся lock record'ы. Туда мы и копируем mark word объекта при легковесной блокировке. Потому, кстати, thin-locked объекты называют stack locked. Раздутый монитор может храниться как у потока, который его раздул, так и в глобальном пуле толстых мониторов.

Директива synchronized

Синхронизация достигается в Java использованием зарезервированного слова synchronized. Вы можете использовать его в своих классах определяя синхронизированные методы или блоки. Вы не сможете использовать synchronized в переменных или атрибутах в определении класса.

Блокировка на уровне объекта

Это механизм синхронизации не статического метода или не статического блока кода, такой, что только один поток сможет выполнить данный блок или метод на данном экземпляре класса.
Это нужно делать всегда, когда необходимо сделать данные на уровне экземпляра потокобезопасными.
Пример:

```
public class DemoClass{
   public synchronized void demoMethod(){}
}
```

или

```
public class DemoClass{
   public void demoMethod(){
      synchronized (this)       {
         //other thread safe code
      }
   }
}
```

или

```
public class DemoClass{
   private final Object lock = new Object();
   public void demoMethod(){
      synchronized (lock)       {
         //other thread safe code
      }
   }
}
```

Блокировка на уровне класса

Предотвращает возможность нескольким потокам войти в синхронизированный блок во время выполнения в любом из доступных экземпляров класса. Это означает, что если во время выполнения программы имеется 100 экземпляров класса DemoClass, то только один поток в это время сможет выполнить demoMethod() в любом из случаев, и все другие случаи будут заблокированы для других потоков.
Это необходимо когда требуется сделать статические данные потокобезопасными.

```
public class DemoClass{
   public synchronized static void demoMethod(){}
}
```

или

```
public class DemoClass{
   public void demoMethod(){
      synchronized (DemoClass.class)       {
         //other thread safe code
      }
   }
}
```

или

```
public class DemoClass
{
   private final static Object lock = new Object();
   public void demoMethod(){
      synchronized (lock)       {
         //other thread safe code
      }
   }
}
```

Дополнительные особенности

Синхронизация в Java гарантирует, что никакие два потока не смогут выполнить синхронизированный метод одновременно или параллельно. synchronized можно использовать только с методами и блоками кода. Эти методы или блоки могут быть статическими или не-статическими.

Когда какой либо поток входит в синхронизированный метод или блок он приобретает блокировку и всякий раз, когда поток выходит из синхронизированного метода или блока JVM снимает блокировку. Блокировка снимается, даже если нить оставляет синхронизированный метод после завершения из-за каких-либо ошибок или исключений. synchronized в Java реентерабельна это означает, что если синхронизированный метод вызывает другой синхронизированный метод, который требует такой же замок, то текущий поток, который держит замок может войти в этот метод не приобретая замок. Синхронизация в Java будет бросать NullPointerException если объект используемый в синхронизированном блоке null. Например, в вышеприведенном примере кода, если замок инициализируется как null, синхронизированный (lock) бросит NullPointerException.
Синхронизированные методы в Java вносят дополнительные затраты на производительность вашего приложения. Так что используйте синхронизацию, когда она абсолютно необходима. Кроме того, рассмотрите вопрос об использовании синхронизированных блоков кода для синхронизации только критических секций кода. Вполне возможно, что и статический и не статический синхронизированные методы могут работать одновременно или параллельно, потому что они захватывают замок на другой объект.
В соответствии со спецификацией языка вы не можете использовать synchronized в конструкторе это приведет к ошибке компиляции.
Не синхронизируйте по не финальному (no final) полю, потому что ссылка, на не финальное поле может измениться в любое время, а затем другой поток может получить синхронизацию на разных объектах и уже не будет никакой синхронизации вообще. Лучше всего использовать класс String, который уже неизменяемый и финальный.

Wait-notify

Иногда при взаимодействии потоков встает вопрос о извещении одних потоков о действиях других. Например, действия одного потока зависят от результата действий другого потока, и надо как-то известить один поток, что второй поток произвел некую работу. И для подобных ситуаций у класса Object определено ряд методов:

- wait(): освобождает монитор и переводит вызывающий поток в состояние ожидания до тех пор, пока другой поток не вызовет метод notify()

- notify(): продолжает работу потока, у которого ранее был вызван метод wait()
- notifyAll(): возобновляет работу всех потоков, у которых ранее был вызван метод wait()

Все эти методы вызываются только из синхронизированного контекста - синхронизированного блока или метода.

Рассмотрим, как мы можем использовать эти методы. Возьмем стандартную задачу из прошлой темы - "Производитель-Потребитель" ("Producer-Consumer"): пока производитель не произвел продукт, потребитель не может его купить. Пусть производитель должен произвести 5 товаров, соответственно потребитель должен их все купить. Но при этом одновременно на складе может находиться не более 3 товаров. Для решения этой задачи задействуем методы wait() и notify():

```
public class ThreadsApp {
    public static void main(String[] args) {
        Store store=new Store();
        Producer producer = new Producer(store);
        Consumer consumer = new Consumer(store);
        new Thread(producer).start();
        new Thread(consumer).start();
    }
}

// Класс Магазин, хранящий произведенные товары
class Store{
   private int product=0;
   public synchronized void get() {
      while (product<1) {
         try {
            wait();
         }
         catch (InterruptedException e) {
         }
      }
      product--;
      System.out.println("Покупатель купил 1 товар");
      System.out.println("Товаров на складе: " + product);
      notify();
   }
   public synchronized void put() {
       while (product>=3) {
         try {
            wait();
         }
         catch (InterruptedException e) {
         }
      }
      product++;
      System.out.println("Производитель добавил 1 товар");
      System.out.println("Товаров на складе: " + product);
      notify();
   }
}




// класс Производитель
class Producer implements Runnable{
    Store store;
    Producer(Store store){
```

```
        this.store=store;
    }
    public void run(){
        for (int i = 1; i < 6; i++) {
            store.put();
        }
    }
}
// Класс Потребитель
class Consumer implements Runnable{
     Store store;
    Consumer(Store store){
        this.store=store;
    }
    public void run(){
        for (int i = 1; i < 6; i++) {
            store.get();
        }
    }
}
```

Итак, здесь определен класс магазина, потребителя и покупателя. Производитель в методе run() добавляет в объект Store с помощью его метода put() 6 товаров. Потребитель в методе run() в цикле обращается к методу get объекта Store для получения этих товаров. Оба метода Store - put и get являются синхронизированными.
Для отслеживания наличия товаров в классе Store проверяем значение переменной product. По умолчанию товара нет, поэтому переменная равна 0. Метод get() - получение товара должен срабатывать только при наличии хотя бы одного товара. Поэтому в методе get в бесконечном цикле проверяем, отсутствует ли товар:

```
while (product<1)
```

Если товар отсутсвует, вызывается метод wait(). Этот метод освобождает монитор объекта Store и блокирует выполнение метода get, пока для этого же монитора не будет вызван метод notify().
Когда в методе put() добавляется товар и вызывается notify(), то метод get() получает монитор и выходит из цикла while (product<1), так как товар добавлен. Затем имитируется получение покупателем товара. Для этого выводится сообщение, и уменьшается значение product: product--. И в конце вызов метода notify() дает сигнал методу put() продолжить работу.
В методе put() работает похожая логика, только теперь метод put() должен срабатывать, если в магазине не более трех товаров. Поэтому в цикле проверяется наличие товара, и если товар уже есть, то освобождаем монитор с помощью wait() и ждем вызова notify() в методе get().
Задание для самостоятельного разбора:
Какой результат выведет программа выше? Ответить необходимо без запуска кода.

Директива JOIN

В Java предусмотрен механизм, позволяющий одному потоку ждать завершения выполнения другого. Для этого используется метод join(). Например, чтобы главный поток подождал завершения побочного потока myThready, необходимо выполнить инструкцию myThready.join() в главном потоке. Как только поток myThready завершится, метод join() вернет управление, и главный поток сможет продолжить выполнение.

Метод join() имеет перегруженную версию, которая получает в качестве параметра время ожидания. В этом случае join() возвращает управление либо когда завершится ожидаемый поток, либо когда закончится время ожидания. Подобно методу Thread.sleep() метод join может ждать в течение миллисекунд и наносекунд – аргументы те же.

С помощью задания времени ожидания потока можно, например, выполнять обновление анимированной картинки пока главный (или любой другой) поток ждёт завершения побочного потока, выполняющего ресурсоёмкие операции:

```
Thinker brain = new Thinker();       //Thinker - потомок класса Thread.
brain.start();            //Начать "обдумывание".
do
{
        mThinkIndicator.refresh();            //mThinkIndicator - анимированная картинка.
        try{
                brain.join(250);                                    //Подождать окончания мысли
четверть секунды.
        }catch(InterruptedException e){}
}
while(brain.isAlive());
```

В этом примере поток brain (мозг) думает над чем-то, и предполагается, что это занимает у него длительное время. Главный поток ждет его четверть секунды и, в случае, если этого времени на раздумье не хватило, обновляет «индикатор раздумий» (некоторая анимированная картинка). В итоге, во время раздумий, пользователь наблюдает на экране индикатор мыслительного процесса, что дает ему знать, что электронные мозги чем то заняты.

Locks

Синхронизация, это хорошо, но не совершенно. Она имеет некоторые функциональные ограничения - невозможно прервать поток, который ожидает блокировки, так же как невозможно опрашивать или пытаться получить блокировку, не будучи готовым к долгому ожиданию. Синхронизация также требует, чтобы блокировка была снята в том же стековом фрейме, в котором была начата, это правило верно почти все время (и хорошо взаимодействует с обработкой исключительных ситуаций), за исключением небольшого количества случаев, когда блокировка с неблочной структурой может быть большим преимуществом.

Класс reentrantLock

Среда Lock в java.util.concurrent.lock - абстракция для блокировки, допускающая осуществление блокировок, которые реализуются как классы Java, а не как возможность языка. Это дает простор разнообразным вариантам применения Lock, которые могут иметь различные алгоритмы планирования, рабочие характеристики, или семантику блокировки. Класс reentrantLock, который реализует Lock, имеет те же параллелизм и семантику памяти, что и synchronized, но также имеет дополнительные возможности, такие как опрос о блокировании (lock polling), ожидание блокирования заданной длительности и прерываемое ожидание блокировки. Кроме того, он предлагает гораздо более высокую эффективность функционирования в условиях жесткой состязательности. (Другими словами, когда много потоков пытаются получить доступ к ресурсу совместного

использования, JVM потребуется меньше времени на установление очередности потоков и больше времени на ее выполнение.)
Что мы понимаем под блокировкой с повторным входом (reentrant)? Просто то, что есть подсчет сбора данных, связанный с блокировкой, и если поток, который удерживает блокировку, снова ее получает, данные отражают увеличение, и тогда для реального разблокирования нужно два раза снять блокировку. Это аналогично семантике synchronized; если поток входит в синхронный блок, защищенный монитором, который уже принадлежит потоку, потоку будет разрешено дальнейшее функционирование, и блокировка не будет снята, когда поток выйдет из второго (или последующего) блока synchronized, она будет снята только когда он выйдет из первого блока synchronized, в который он вошел под защитой монитора.
Если посмотреть на пример кода в листинге 1, сразу бросается в глаза различие между Lock и синхронизацией - блокировка должна быть снята в последнем блоке. Иначе, если бы защищенный код показал исключительное состояние (ошибку), блокировка не была бы снята! Эта особенность может показаться тривиальной, но, фактически, она очень важна. Если забыть снять блокировку в последнем блоке, это будет бомбой замедленного действия в вашей программе, причину которой будет очень трудно обнаружить, и в конце концов она взорвется. Используя синхронизацию, JVM гарантирует, что блокировка автоматически снимаются.
Листинг 1. Защита блока кода reentrantLock.

```
Lock lock = new reentrantLock();
lock.lock();
try {
  // update object state
}
finally {
  lock.unlock();
}
```

Как дополнительное преимущество можно отметить, что реализация reentrantLock гораздо более масштабируемая в условиях состязательности, чем реализация synchronized. (Вероятно, в состязательном режиме работы синхронных средств будут дальнейшие усовершенствования в следующей версии JVM.) Это значит, что когда много потоков соперничают за право получения блокировки, общая пропускная способность обычно лучше у reentrantLock, чем у synchronized.

Синхронизация данных

Synchronized-коллекции

Первым ассоциативным классом коллекций, который появился библиотеке классов Java, был Hashtable, который являлся частью JDK 1.0. Hashtable предоставлял легкую в использовании, потокобезопасную реализацию ассоциативной map-таблицы и был удобен. Однако потокобезопасность обошлась дорого - все методы в Hashtable были синхронизированы. В то время за синхронизацию без конкуренции приходилось расплачиваться производительностью. Преемник Hashtable, HashMap, появившийся как часть Collections framework в JDK 1.2, решал проблему потокобезопасности, предоставляя несинхронизированный базовый класс и синхронизированное обрамление, Collections.synchronizedMap. Разделение базовой функциональности и потокобезопасности в Collections.synchronizedMap позволило получить синхронизацию

пользователям, которым она была нужна, а тем, кому она не нужна, не приходилось платить за неё цену.
Упрощённый подход к синхронизации, использованный и в Hashtable и в synchronizedMap, - синхронизация каждого метода с Hashtable или с синхронизированным объектом обрамления Map - имеет два основных недостатка. Он является препятствием для масштабируемости, потому что к хеш-таблице может одновременно обращаться только один поток. Одновременно с этим, недостаточно обеспечить настоящую безопасность потоков, при этом множество распространённых составных операций всё ещё требует дополнительной синхронизации. Хотя простые операции вроде get() и put() могут выполняться безопасно без дополнительной синхронизации, существуют несколько распространённых последовательностей операций, таких как iterate или put-if-absent, которые всё же нуждаются во внешней синхронизации, чтобы избежать конкуренции за данные.

Условная потокобезопасность

Синхронизированные обрамления коллекций synchronizedMap и synchronizedList иногда называют условно потокобезопасными- все операции в отдельности потокобезопасны, но последовательности операций, где управляющий поток зависит от результатов предыдущих операций, могут быть причиной конкуренции за данные. Первый отрывок в Листинге 1 демонстрирует распросранённую идиому put-if-absent (запись-если-пусто) - если элемент ещё не существует в Map-таблице, добавить его. К сожалению, как уже говорилось, у другого потока существует возможность вставить значение с повторяющимся ключом между моментами возврата из метода containsKey() и вызова метода put(). Если вы хотите гарантировать, что вставка выполняется строго один раз, вам необходимо заключить пару выражений в синхронизированный блок, который синхронизируется с Map m.
Другие примеры в Листинге 1 связаны с повторением. В первом примере результаты List.size() могли стать недействительными во время выполнения цикла, потому что другой поток мог удалить элементы из списка. Если ситуация сложилась неудачно, и элемент был удален другим потоком сразу же после начала последнего повторения цикла, то List.get() возвратит null и doSomething()скорее всего выбросит NullPointerException. Что вам можно сделать, чтобы избежать этого? Если другой поток может обращаться к List, в то время, когда вы выполняете его перебор, вы должны заблокировать весь List на время перебора, заключив его в блок synchronized, синхронизирующийся с List l. Это решает проблемы с конкуренцией за данные, но в дальнейшем скажется на параллелизме, поскольку блокировка всего List во время перебора может надолго заблокировать доступ к списку другим потокам.
В Collections framework были введены итераторы для обхода списка или другой коллекции, которые оптимизируют процесс перебора элементов коллекции. Однако итераторы, реализованные в классах коллекций java.util, часто подвержены сбоям, что означает, что если один поток изменяет коллекцию в то время, когда другой пропускает её через Iterator, очередной вызов Iterator.hasNext() или Iterator.next() выбросит ConcurrentModificationException. Как и с предыдущим примером, если вы хотите предотвратить ConcurrentModificationException, вам надо целиком блокировать List в то время, когда вы выполняете повторения, заключив его внутрь блока synchronized, который синхронизируется с List l. (В качестве альтернативы, вы можете вызвать List.toArray() и делать перебор в массиве без синхронизации, но это может быть дорого, если список достаточно большой.)

Листинг 1. Типичный случай конкуренции в синхронизированных map-таблицах

```
Map m = Collections.synchronizedMap(new HashMap());
List l = Collections.synchronizedList(new ArrayList());
// put-if-absent idiom - contains a race condition
// may require external synchronization
if (!map.containsKey(key))
  map.put(key, value);
// ad-hoc iteration - contains race conditions
// may require external synchronization
for (int i=0; i<list.size(); i++) {
  doSomething(list.get(i));
}
// normal iteration - can throw ConcurrentModificationException
// may require external synchronization
for (Iterator i=list.iterator(); i.hasNext(); ) {
  doSomething(i.next());
}
```

Ложная потокобезопасность

Условная безопасность потоков, обеспечиваемая synchronizedList и synchronizedMap представляет скрытую угрозу - разработчики полагают, что, раз эти коллекции синхронизированы, значит, они полностью потокобезопасны, и пренебрегают должной синхронизацией составных операций.
В результате, хотя эти программы и работают при лёгкой нагрузке, но при серьёзной нагрузке они могут начать выкидывать NullPointerException или ConcurrentModificationException.

Проблемы масштабируемости

Масштабируемость показывает, как меняется пропускная способность приложения при увеличении рабочей нагрузки и доступных вычислительных ресурсов. Масштабируемая программа может выдержать пропорционально большую нагрузку при большем количестве процессоров, памяти или пропускной способности ввода/вывода. Блокировка ресурса общего пользования ради монопольного доступа представляет собой проблему для масштабируемости - она не позволяет другим потокам получить доступ к этому ресурсу даже при наличии простаивающих процессоров для планирования этих потоков. Для достижения масштабируемости мы должны исключить или сократить нашу зависимость от монопольных блокировок ресурсов.
Ещё большая проблема с синхронизированным обрамлением коллекций и ранними классами Hashtable и Vector состоит в том, что они синхронизируются с единственной блокировкой. Это означает, что одновременно только один поток может иметь доступ к коллекции, и если один поток находится в процессе чтения Map-таблицы, все остальные

потоки, которые хотят прочитать из неё или записать в неё, вынуждены ждать. Наиболее распространённые операции с Map-таблицей, get() и put(), могут требовать больших вычислений, чем это может показаться, - при обходе хеш-бакета в поисках специфического значения ключа get() может потребоваться вызов Object.equals() для большого количества кандидатов. Если функция hashCode(), используемая классом ключа, не распределяет значения равномерно по всему ряду хэшей или же имеет большое количество коллизий хэша, отдельные цепочки бакетов могут оказаться намного длиннее других, а прохождение длинной цепочки хэшей и вызов equals() на определённом проценте его элементов могут быть слишком медленными. Проблема высокой стоимости get() и put() при таких условиях выражается не только в том, что доступ будет медленным, но и в том, что всем другим потокам блокируется доступ кMap-таблице пока происходит обход этой цепочки хэшей.
Тот факт, что get() в некоторых случаях может требовать значительного времени на выполнение, ещё больше усугубляется проблемой условной безопасности потоков, обсуждавшейся выше.
Одно из наиболее распространённых применений для Map в серверных приложениях - реализация кэш-памяти. Серверные приложения могут кэшировать содержимое файлов, сгенерированные страницы, результаты запросов к базам данных, деревья DOM, ассоциированные с анализируемыми XML-файлами и многие другие типы данных. Основное назначение кэш-памяти - сокращение времени обслуживания и увеличение пропускной способности за счёт использования результатов предыдущего вычисления. Типичная особенность рабочей нагрузки кэш-памяти состоит в том, что попадания встречаются гораздо чаще, чем обновления, поэтому (в идеале) кэш-память обеспечивает очень хорошую производительность для get(). Кэш-память, тормозящая производительность приложения, даже хуже, чем полное отсутствие кэш-памяти.
Если вы использовали synchronizedMap для реализации кэш-памяти, вы вносите в ваше приложение потенциальную проблему масштабируемости. Одновременно только один поток может иметь доступ к Map, и это распространяется на все потоки, которые могут извлекать значение из Map-таблицы, равно как и на потоки, которые желают вставить новую пару (key, value) в map-таблицу.

Сокращение размеров блокировок

Один из подходов к улучшению параллелизма HashMap при сохранении потокобезопасности состоит в том, чтобы обходиться без единой блокировки всей таблицы и использовать блокировки для каждого бакета хэшей (или, в более общем случае, пула блокировок, где каждая блокировка защищает несколько бакетов). Это означает, что сразу несколько потоков могут обращаться к различным частям Map одновременно, без соперничества за единственную на всю коллекцию блокировку. Этот подход сразу же улучшает масштабируемость операций вставки, извлечения и удаления. К сожалению, такой параллелизм имеет свою цену - сложнее становится реализовывать методы, работающие с целой коллекцией, такие как size() или isEmpty(), потому что для этого может потребоваться получение сразу множества блокировок или появляется риск вернуть неточный результат. Тем не менее, для ситуаций вроде реализации кэш-памяти это представляет разумный компромисс - операции извлечения и вставки часты, тогда как операции size() и isEmpty() - значитель менее частые.

ConcurrentHashMap

Класс ConcurrentHashMap из util.concurrent (который также появится в пакете java.util.concurrent в JDK 1.5) - это потокобезопасная реализация Map, предоставляющая намного большую степень параллелизма, чем synchronizedMap. Сразу много операций чтения могут почти всегда выполняться параллельно, одновременные чтения и записи могут обычно выполняться параллельно, а сразу несколько одновременных записей могут зачастую выполняться параллельно. (Соответственный класс ConcurrentReaderHashMap предлагает аналогичный параллелизм для множественных операций чтений, но допускает лишь одну активную операцию записи.) ConcurrentHashMap спроектирован для оптимизации операций извлечения; на деле, успешные операции get() обычно успешно выполняются безо всяких блокировок. Достижение потокобезопасности без блокировок является сложным и требует глубокого понимания деталей Модели Памяти Java (Java Memory Model). Реализация ConcurrentHashMap и остальная часть util.concurrent были в значительной степени проанализированы экспертами по параллелизму на предмет корректности и безопасности потоков. Мы рассмотрим детали реализации ConcurrentHashMap в статье в следующем месяце.

ConcurrentHashMap добивается более высокой степени параллелизма, слегка смягчая обещания, которые даются тем, кто её вызывает. Операция извлечения возвратит значение, вставленное самой последней завершившейся операцией вставки, а также может возвратить значение, добавленное операцией вставки, выполняемой в настоящее время (но она никогда не возвратит бессмыслицы). Итераторы, возвращаемые ConcurrentHashMap.iterator() возвратят каждый элемент не более одного раза и никогда не выкинут ConcurrentModificationException, но могут отображать или не отображать вставки или удаления, имевшие место со времени, когда итератор был сконструирован. Блокировки целой таблицы не требуются (да и невозможны) для обеспечения потокобезопасности при переборе коллекции. ConcurrentHashMap может использоваться для замены synchronizedMap или Hashtable в любом приложении, которое не основано на способности делать блокировку всей таблицы для предотвращения модификаций.

Данные компромиссы позволяют ConcurrentHashMap обеспечивать намного более высокую масштабируемость, чем Hashtable, не ставя под угрозу его эффективность для широкого множества распространённых случаев, таких как кэш-память с общим доступом.

CopyOnWriteArrayList

Класс CopyOnWriteArrayList предназначен на замену ArrayList в параллельных приложениях, где обходы значительно превосходят по количеству вставки и удаления. Это достаточно типично для случая, когда ArrayList используется для хранения списка подписчиков, как в приложениях AWT или Swing или вообще в классах JavaBean. (Похожие на них CopyOnWriteArraySetиспользуют CopyOnWriteArrayList для реализации интерфейса Set.)

Если вы используете обычный ArrayList для хранения списка подписчиков, то до тех пор, пока список допускает изменения и к нему могут обращаться много потоков, вы должны либо блокировать целый список во время перебора либо клонировать его перед перебором, оба варианта обходятся значительной ценой. CopyOnWriteArrayList вместо этого создаёт новую копию списка каждый раз, когда выполняется модифицирующая операция, и гарантируется, что её итераторы возвращают состояние списка на момент, когда итератор был сконструирован и не выкинут ConcurrentModificationException. Нет необходимости клонировать список до перебора или блокировать его во время перебора, потому что копия списка, которую видит итератор, не будет изменяться. Другими словами, CopyOnWriteArrayList содержит изменяемую ссылку на неизменяемый массив,

поэтому до тех пор, пока эта ссылка остаётся фиксированной, вы получаете все преимущества потокобезопасности от неизменности без необходимости блокировок.
Синхронизированные классы коллекций Hashtable и Vector и синхронизированные классы обрамления Collections.synchronizedMap и Collections.synchronizedList предоставляют базовую условно потокобезопасную реализацию Map и List. Однако несколько факторов делают их непригодными для использования в приложениях с высоким уровнем параллелизма - используемая в них единственная блокировка на всю коллекцию является препятствием для масштабируемости, а зачастую бывает необходимо блокировать коллекцию на значительное время в момент перебора для предотвращения исключений ConcurrentModificationException.
Реализации ConcurrentHashMap и CopyOnWriteArrayList обеспечивают намного больший уровень параллелизма при сохранении безопасности потоков и при нескольких незначительных компромиссах в их обещаниях перед вызывающей стороной. ConcurrentHashMap и CopyOnWriteArrayList не обязательно полезны везде, где вы могли бы использовать HashMap или ArrayList, но они спроектированы для оптимизации в определённых распространённых ситуациях. Многие параллельные приложения будут в выигрыше от их использования.

Atomic-типы

Мы уже говорили, что большинство современных процессоров имеют поддержку многопроцессорной обработки. Это означает, что несколько процессоров могут совместно пользоваться периферийными устройствами и ОЗУ, и кроме этого имеет такие дополнения к набору команд, которые обеспечивают многопроцессорную обработку. В частности, почти все новые процессоры имеют команды для обновления переменных совместного доступа таким образом, чтобы распознать или предотвратить параллельный доступ разными процессорами.

Сравнение и замена (Compare and swap - CAS)

Первые процессоры, поддерживавшие параллелизм в обработке, обеспечивали атомарные операции команд установки семафора (test-and-set), которые обычно применялись к отдельному биту. Наиболее частый подход в современных процессорах, включая Intel и Sparc, - использование базового элемента под названием Сравнение и замена (CAS) (В процессорах Intel он осуществляется группой команд cmpxchg. Процессоры PowerPC имеют 2 команды: load and reserve (загрузить и резервировать) и store conditional (сохранить при условии); они предназначены для выполнения той же задачи, то же можно сказать о MIPS-процессорах, хотя в них первая носит название load linked (загрузить со связкой)).
Операция CAS включает 3 объекта-операнда: адрес ячейки памяти (V), ожидаемое старое значение (A) и новое значение (B). Процессор атомарно обновляет адрес ячейки, если новое значение совпадает со старым, иначе изменений не зафиксируется. В любом случае, будет выведена величина, которая предшествовала времени запроса. Некоторые варианты метода CAS просто сообщают, успешно ли прошла операция, вместо того, чтобы отобразить само текущее значение. Фактически, CAS только сообщает: "Наверное, адрес V равняется A; если так оно и есть, поместите туда же B, в противном случае не делайте этого, но обязательно скажите мне, какая величина - текущая."
Самым естественным методом использования CAS для синхронизации будет чтение значения A с адреса V, проделать многошаговое вычисление для получения нового значения B, и затем воспользоваться методом CAS для замены значения параметра V с прежнего, A, на новое, B. CAS выполнит задание, если V за это время не менялось.

Задания таких методов, как CAS, позволяют алгоритму выполнить последовательность операций "чтение-модификация-запись" без опасения, что в это же время другой поток изменит переменную, потому что если бы это произошло, CAS обнаружил бы это и прервал бы выполнение своего задания, а алгоритм задал бы повтор операции. Листинг 3 показывает ход выполнения CAS (не отображая его характеристик), но величина, выводимая CAS, соответствует установкам оборудования и очень небольшая (для большинства процессоров):

Листинг 3. Код, иллюстрирующий ход выполнения (но не характеристики) метода Сравнить и заменить

```
public class SimulatedCAS {
    private int value;
public synchronized int getValue() { return value; }
public synchronized int compareAndSwap(int expectedValue, int newValue) {
        int oldValue = value;
        if (value == expectedValue)
           value = newValue;
        return oldValue;
    }
}
```

Реализация счетчиков методом CAS

Параллельные алгоритмы на основе CAS называют безблокировочными, потому что потокам не приходится ожидать блокировки (иногда ее называют разделом совместного доступа (mutex) или критическим разделом, в зависимости от того, какой терминологией пользуется ваша потоковая платформа). Операция CAS проходит успешно или неуспешно, но независимо от исхода она завершается в предсказуемые сроки. Если CAS не реализуется, вызывающая программа может повторить попытку операции CAS или предпринять иные шаги, которые она считает нужными. Листинг 4 показывает класс счетчика, переписанный для использования CAS вместо блокировки:

Листинг 4. Реализация счетчика с методом Сравнение и замена

```
public class CasCounter {
   private SimulatedCAS value;
   public int getValue() {
      return value.getValue();
   }
   public int increment() {
      int oldValue = value.getValue();
      while (value.compareAndSwap(oldValue, oldValue + 1) != oldValue)
         oldValue = value.getValue();
      return oldValue + 1;
   }
}
```

Безблокировочные алгоритмы и алгоритмы, не требующие ожидания (wait-free)

Алгоритм называют wait-free, если каждый поток будет продолжать обработку при наличии произвольной задержки (arbitrary delay) или сбоя других потоков. Напротив, алгоритм lock-free требует только того, чтобы работа постоянно шла хотя бы

в одном потоке шла обработка. (Еще один способ дать определение wait-free: при этом методе каждый поток гарантированно правильно производит вычисления по своим операциям в ограниченном количестве своих шагов, независимо от действий, хронометража, чередования (interleaving) и скорости остальных потоков. Это ограничение может накладываться функцией количества потоков в системе; например, если каждый из 10 потоков выполнит по одному разу операцию CasCounter.increment(), то самое худшее, что может случиться, это дополнительные 9 попыток со стороны каждого потока до того, как завершится обновление значения (increment).)

Алгоритмам wait-free и безблокировочным алгоритмам (lock-free, nonblocking) за последние 15 лет было посвящено много исследований, и были открыты безблокировочные механизмы для многих структур с общими данными. Безблокировочные алгоритмы широко используют на уровне операционной системы и в виртуальной машине Java для решения задач планирования потоков и процессов. Хотя их труднее реализовывать, они обладают несколькими преимуществами по сравнению с блокировочными методами - исчезает опасность смены приоритетов и взаимного блокирования, уменьшаются затраты на разрешение конфликтов и координация происходит на уровне более мелких ячеек, что обеспечивает более высокий уровень параллелизма.

Классы атомарных переменных

До версии JDK 5.0 на языке Java нельзя было создавать алгоритмы wait-free и безблокировочные без использования собственного кода. С появлением классов атомарных переменных в пакете java.util.concurrent.atomic все изменилось. Все атомарные классы переменных имеют базовый элемент Сравнение и назначение (compare-and-set) (аналогичный элементу Сравнение и замена), который реализуется при помощи самого быстрого собственного структурного компонента, кооторый имеется в платформе (Сравнение и замена, Загрузить в связке, Сохранить при условии или в крайнем случае спин-блокировками). В пакет java.util.concurrent.atomic входят 9 видов атомарных переменных (AtomicInteger; AtomicLong; AtomicReference; AtomicBoolean; формы для массивов атомарных целых чисел; длинные (long); ссылки; а также атомарные с пометкой Класс эталона (reference), которые атомарно обновляют две величины).

Классы атомарных переменных можно рассматривать как обобщение volatile переменных, если расширить понятие изменяемых переменных до переменных с поддержкой атомарных обновлений методом Сравнение и назначение. Чтение и запись атомарных переменных имеет такую же семантику памяти как доступ к чтению и записи изменяемых переменных.

Хотя классы атомарных переменных могут показаться похожими на пример кода SynchronizedCounter в Листинге 1, сходство только поверхностное. Под поверхностью операции с атомарными переменными превращаются в аппаратные базовые элементы, которые платформа предоставляет для параллельного доступа, например, в методе Сравнение и замена.

Мелкоячеистость означает отсутствие тяжеловесности

Привычным приемом для настройки масштабируемости параллельного приложения, в котором происходит конфликт доступа, является уменьшение размера ячеек блокируемых объектов; его применяют с надеждой на то, что некоторые блокировки перестанут конфликтовать. Переход с блокировки на атомарные переменные приводит к тому же результату - потому что мы осуществляемом мелкоячеистость механизма координации и меньше операций оказываются в ситуации конфликта, что повышает пропускную способность.

Атомарные переменные в java.util.concurrent

Почти все классы в пакете java.util.concurrent используют атомарные переменные вместо синхронизации, либо прямо, либо косвенно. Такие классы, как ConcurrentLinkedQueue используют атомарные переменные для прямой реализации алгоритмов wait-free и такие классы, как ConcurrentHashMap используют ReentrantLock для блокировки, если это необходимо. В свою очередь, ReentrantLock использует атомарные переменные для соблюдения очередности потоков, ожидающих блокировки.
Эти классы невозможно было бы сконструировать без усовершенствования машины JVM в версии JDK 5.0, которая сделала явным (для библиотек классов, но не для библиотек пользователей) интерфейс доступа к базовым элементам синхронизации аппаратного уровня. Классы атомарных переменных и, в свою очередь, другие классы в java.util.concurrent экспонируют эти характеристики классам пользователей.

Повышение пропускной способности при помощи атомарных переменных

Добавим еще одну реализацию к этому тесту, которая использует AtomicLong для обновления состояния генератора псевдослучайных чисел (PRNG).
Листинг 5 показывает реализацию PRNG при помощи синхронизации, а также альтернативную ей реализацию с CAS. Обратите внимание на то, что CAS надо запускать циклично, потому что успешной реализации могут предшествовать сбои, что типично для кодов, использующих CAS.
Листинг 5. Реализация PRNG с защищённым потоком, синхронизацией и атомарными переменными

```
public class PseudoRandomUsingSynch implements PseudoRandom {
   private int seed;
   public PseudoRandomUsingSynch(int s) { seed = s; }
   public synchronized int nextInt(int n) {
      int s = seed;
      seed = Util.calculateNext(seed);
      return s % n;
   }
}

public class PseudoRandomUsingAtomic implements PseudoRandom {
   private final AtomicInteger seed;
   public PseudoRandomUsingAtomic(int s) {
      seed = new AtomicInteger(s);
   }
   public int nextInt(int n) {
      for (;;) {
         int s = seed.get();
         int nexts = Util.calculateNext(s);
         if (seed.compareAndSet(s, nexts))
            return s % n;
      }
   }
}
```

Графики 1 и 2 показывают пропускную способность (в прокрутках в секунду) при генерации случайных чисел с помощью различного количества потоков на 8-канальной машине Ultrasparc3 и на однопроцессорной машине Pentium 4. Количество потоков в этих тестах обманчиво, эти потоки обнаруживают гораздо больше конфликтов, чем обычно, поэтому они показывают коэффициент доходной загрузки (break-even)

между ReentrantLock и атомарными переменными при гораздо меньшем количестве потоков, чем это выглядело бы в более реальной программе. Вы видите, что атомарные переменные предлагают дополнительное усовершенствование по сравнению с ReentrantLock, который и без того имел то преимущество перед синхронизацией. (Поскольку на каждом этапе работы выполняется столь мало, нижеприведенные графики, возможно, занижают реальную выгоду, получаемую благодаря увеличению масштабируемости, которое появилось при замене блока ReentrantLock на атомарные переменные.)

Рисунок 1. Тест пропускной способности при синхронизации, блоке ReentrantLock, справедливом блоке и AtomicLong на 8-канальном Ultrasparc3

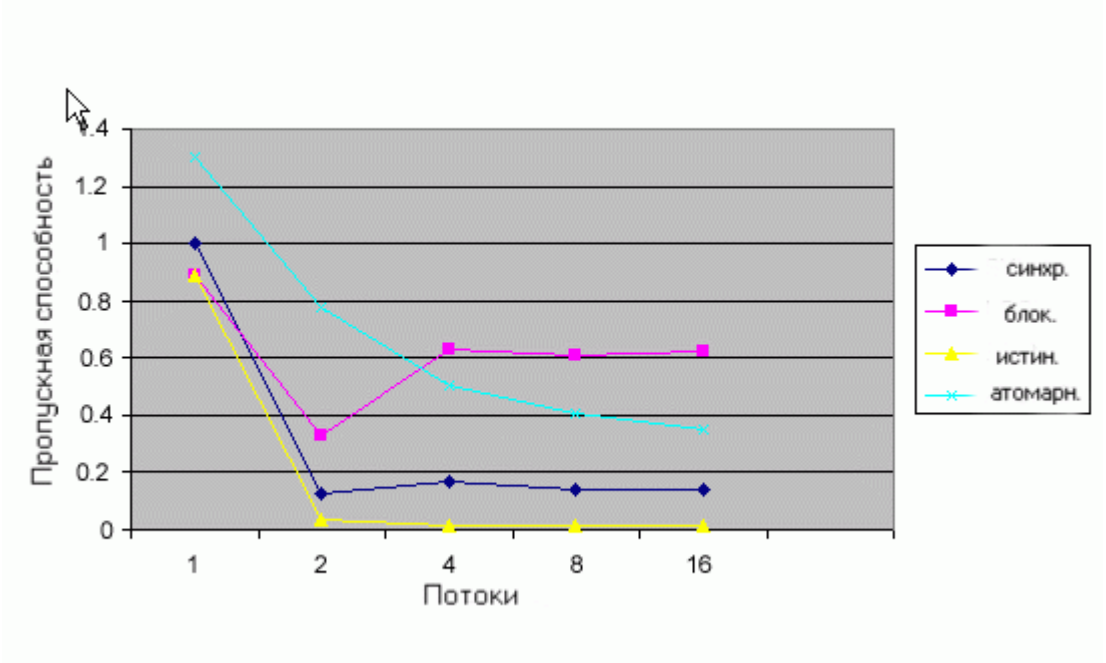
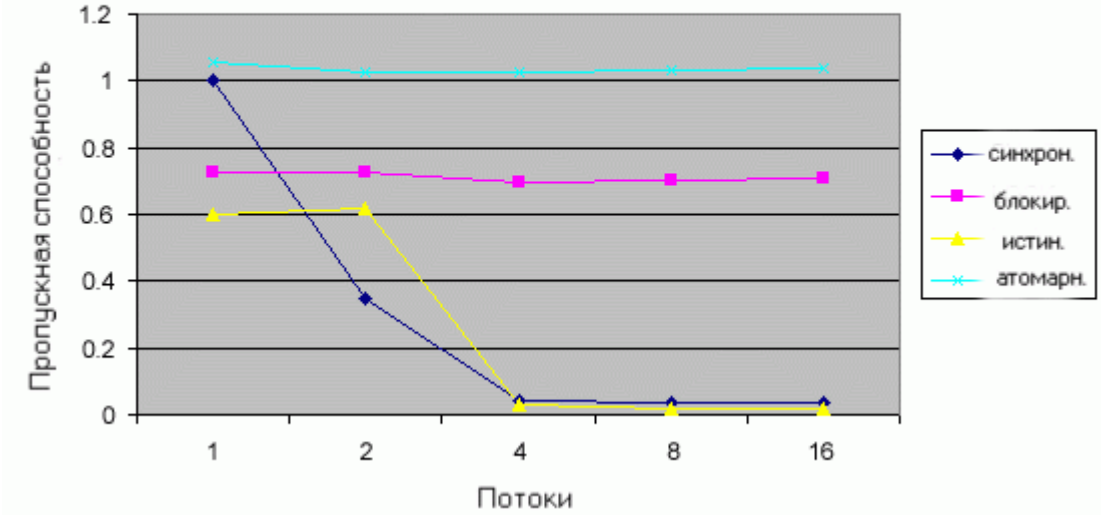


Рисунок 2. Тест пропускной способности при синхронизации, блоке ReentrantLock, справедливом блоке и AtomicLong на однопроцессорном Pentium 4



Едва ли многие пользователи станут самостоятельно разрабатывать безблокировочные алгоритмы с атомарными переменными - скорее всего, они используют версии, предоставляемые в java.util.concurrent, такие, где повышение продуктивности является результатом ConcurrentLinkedQueue. Но если вам все-таки хочется узнать, за счет чего так возрастает производительность по сравнению с аналогами в предыдущих версиях JDK,

знайте: все дело в том, что используются мелкоячеистые аппаратные базовые элементы, выраженные через классы атомарных переменных.

Deadlock

Рассмотрим следующую задачу: необходимо написать метод осуществляющий транзакцию перевода некоторого количества денег с одного счета на другой. Решение может иметь следующий вид:

```
public void transferMoney(Account fromAccount, Account toAccount, Amount amount)
throws InsufficientFundsException {
        synchronized (fromAccount) {
                synchronized (toAccount) {
                        if (fromAccount.getBalance().compareTo(amount) < 0) throw new
InsufficientFundsException();
                        else {  fromAccount.debit(amount); toAccount.credit(amount);  }
                }
        }
}
```

На первый взгляд, данный код синхронизирован вполне нормально, мы имеем атомарную операцию проверки и изменения состояния счета-источника и изменение счета-получателя. Но, при данной стратегии синхронизации может возникнуть ситуация взаимной блокировки. Давайте рассмотрим пример того, как это происходит. Необходимо произвести две транзакции: со счета A на счет B перевести x денег, а со счета B на счет A – y. Зачастую эта ситуация не вызовет взаимной блокировки, однако, при неудачном стечении обстоятельств, транзакция 1 займет монитор счета A, транзакция 2 займет монитор счета B. Результат – взаимная блокировка: транзакция 1 ждет, пока транзакция 2 освободит монитор счета B, но для этого транзакция 2 должна получить доступ к монитору A, занятому транзакцией 1.
Одна из больших проблем с взаимными блокировками – что их нелегко найти при тестировании. Даже в ситуации, описанной в примере, потоки могут не заблокироваться, то есть данная ситуация не будет постоянно воспроизводится, что значительно усложняет диагностику. В целом описанная проблема недетерминированности является типичной для многопоточности (хотя от этого не легче). Потому, в повышении качества многопоточных приложений важную роль играет code review, поскольку он позволяет выявить ошибки, которые проблематично воспроизвести при тестировании. Это не значит, что приложение не надо тестировать, просто о code review тоже не надо забывать.
Что нужно сделать, чтобы этот код не приводил к взаимной блокировке? Данная блокировка вызвана тем, что синхронизация счетов может происходить в разном порядке. Соответственно, если ввести некоторый порядок на счетах (это некоторое правило, позволяющее сказать, что счет A меньше чем счет B), то проблема будет устранена. Как это сделать? Во-первых, если у счетов есть какой-то уникальный идентификатор (например, номер счета) численный, строчный или еще какой-то с естественным понятием порядка (строки можно сравнивать в лексикографическом порядке, то можем считать, что нам повезло, и мы всегда можем сначала занимать монитор меньшего счета, а потом большего (или наоборот).

```
private void doTransfer(final Account fromAcct, final Account toAcct, final DollarAmount
amount) throws InsufficientFundsException {
        if (fromAcct.getBalance().compareTo(amount) < 0)
                throw new InsufficientFundsException();
        else {
                fromAcct.debit(amount);
                toAcct.credit(amount);
        }
}
public void transferMoney(final Account fromAcct, final Account toAcct, final DollarAmount
amount) throws InsufficientFundsException {
        int fromId= fromAcct.getId();
        int toId = fromAcct.getId();
        if (fromId < toId) {
                synchronized (fromAcct) {
                        synchronized (toAcct) {
                                doTransfer(fromAcct, toAcct, amount)}
                        }
                }
        } else  {
                synchronized (toAcct) {
                        synchronized (fromAcct) {
                                doTransfer(fromAcct, toAcct, amount)}
                        }
                }
        }
}
```

Второй вариант, если такого идентификатора у нас нет, то придется его придумать самим. Мы можем в первом приближении сравнивать объекты по хеш-коду. Скорее всего, они будут отличаться. Но что делать, если они все же окажутся одинаковыми? Тогда придется добавить еще один объект для синхронизации. Это может выглядеть несколько изощренным, но что поделать. Да и к тому же, третий объект будет использоваться довольно редко. Результат будет выглядеть следующим образом:

```
private static final Object tieLock = new Object();
private void doTransfer(final Account fromAcct, final Account toAcct, final DollarAmount
amount) throws InsufficientFundsException {
        if (fromAcct.getBalance().compareTo(amount) < 0)
                throw new InsufficientFundsException();
        else {
                fromAcct.debit(amount);
                toAcct.credit(amount);
        }
}
public void transferMoney(final Account fromAcct, final Account toAcct, final DollarAmount
amount) throws InsufficientFundsException {
        int fromHash = System.identityHashCode(fromAcct);
        int toHash = System.identityHashCode(toAcct);
        if (fromHash < toHash) {
                synchronized (fromAcct) {
                        synchronized (toAcct) {
                                doTransfer(fromAcct, toAcct, amount);
                        }
                }
        } else if (fromHash > toHash) {
                synchronized (toAcct) {
                        synchronized (fromAcct) {
                                doTransfer(fromAcct, toAcct, amount);
                        }
```

```
                }
        } else {
                synchronized (tieLock) {
                        synchronized (fromAcct) {
                                synchronized (toAcct) {
                                        doTransfer(fromAcct, toAcct, amount)
                                }
                        }
                }
        }
}
```

Взаимная блокировка между объектами

Описанные условия блокировки представляют наиболее простой по диагностике случай взаимной блокировки. Зачастую в многопоточных приложениях различные объекты пытаются получить доступ к одним и тем же синхронизированным блокам. При этом может возникнуть взаимная блокировка. Рассмотрим следующий пример: приложение для диспетчера полетов. Самолеты сообщают диспетчеру, когда они прибыли на место назначения и запрашивают разрешение на посадку. Диспетчер хранит всю информацию о самолетах, летящих в его направлении, и может строить их положение на карте.

```
class Plane {
        private Point location, destination;
        private final Dispatcher dispatcher;
        public Plane(Dispatcher dispatcher) {
                this.dispatcher = dispatcher;
        }
        public synchronized Point getLocation() {
                return location;
        }
        public synchronized void setLocation(Point location) {
                this.location = location;
                if (location.equals(destination))
                dispatcher.requestLanding(this);
        }
}

class Dispatcher {
        private final Set<Plane> planes;
        private final Set< Plane > planesPendingLanding;
        public Dispatcher() {
                planes = new HashSet<Plane>();
                planesPendingLanding = new HashSet<Plane>();
        }
        public synchronized void requestLanding(Plane plane) {
                planesPendingLanding.add(plane);
        }
        public synchronized Image getMap() {
                Image image = new Image();
                for (Plane plane : planes)
                        image.drawMarker(plane.getLocation());
                return image;
        }
}
```

Понять, что в этом код есть ошибка, которая может привести к взаимной блокировке сложнее, чем в предыдущем. На первый взгляд, в нем нет повторных синхронизаций, однако это не так. Вы, наверное, уже заметили, что методы setLocation класса Plane и

getMap класса Dispatcher, являются синхронизированными и вызывают внутри себя синхронизированные методы других классов. Это в целом плохая практика. О том, как это можно исправить, речь пойдет в следующем разделе. В результате, если самолет прибывает на место, в тот же момент, как кто-то решает получить карту может возникнуть взаимная блокировка. То есть, будут вызваны методы, getMap и setLocation, которые займут мониторы экземпляров Dispatcher и Plane соответственно. Затем метод getMap вызовет plane.getLocation (в частности для экземпляра Plane, который в данный момент занят), который будет ждать освобождения монитора для каждого из экземпляров Plane. В то же время в методе setLocation будет вызван dispatcher.requestLanding, при этом монитор экземпляра Dispatcher остается занят рисованием карты. Результат – взаимная блокировка.

Модель памяти JAVA

Модель памяти Java описывает поведение потоков в среде исполнения Java. Модель памяти — часть семантики языка Java, и описывает, на что может и на что не должен рассчитывать программист, разрабатывающий ПО не для конкретной Java-машины, а для Java в целом.

Atomicity

Хотя многие это знают, считаю необходимым напомнить, что на некоторых платформах некоторые операции записи могут оказаться неатомарными. То есть, пока идёт запись значения одним потоком, другой поток может увидеть какое-то промежуточное состояние. За примером далеко ходить не нужно — записи тех же long и double, если они не объявлены как volatile, не обязаны быть атомарными и на многих платформах записываются в две операции: старшие и младшие 32 бита отдельно.

Visibility

В старой JMM у каждого из запущенных потоков был свой кеш (working memory), в котором хранились некоторые состояния объектов, которыми этот поток манипулировал. При некоторых условиях кеш синхронизировался с основной памятью (main memory), но тем не менее существенную часть времени значения в основной памяти и в кеше могли расходиться.
В новой модели памяти от такой концепции отказались, потому что то, где именно хранится значение, вообще никому не интересно. Важно лишь то, при каких условиях один поток видит изменения, выполненные другим потоком. Кроме того, железо и без того достаточно умно, чтобы что-то кешировать, складывать в регистры и вытворять прочие операции.
Важно отметить, что, в отличие от того же C++, «из воздуха» (out-of-thin-air) значения никогда не берутся: для любой переменной справедливо, что значение, наблюдаемое потоком, либо было ранее ей присвоено, либо является значением по умолчанию.

Reordering

Также, инструкции в потоке могут быть переставлены местами, если не нарушают программный порядок. Одним из примечательных эффектов может оказаться то, что

действия, выполненные одним потоком, другой поток увидит в другом порядке. Эту фразу довольно сложно понять, просто прочитав, потому приведу пример. Пусть есть такой код:

```
public class ReorderingSample {
   boolean first = false;
   boolean second = false;
   void setValues() {
      first = true;
      second = true;
   }
   void checkValues() {
      while(!second);
      assert first;
   }
}
```

И в этом коде из одного потока вызывается метод checkValues, а из другого потока — setValues. Казалось бы, код должен выполняться без проблем, ведь полю second значение true присваивается позже, чем полю first, и потому когда (точнее, если) мы видим, что, второе поле истинно, то и первое тоже должно быть таким.
Хотя внутри одного потока об этом можно не беспокоиться, в многопоточной среде результаты операций, произведённых другими потоками, могут наблюдаться не в том порядке
Пусть класс Data в конструкторе выполняет какие-то не очень тривиальные вычисления и, главное, записывает какие-то значения в не final поля:

```
public class Data {
   String question;
   int answer;
   int maxAllowedValue;
   public Data() {
      this.answer = 42;
      this.question = reverseEngineer(this.answer);
      this.maxAllowedValue = 9000;
   }
}
```

Получится, что тот поток, который первый обнаружит, что data == null, выполнит следующие действия:

1. Выделит память под новый объект
2. Вызовет конструктор класса Data
3. Запишет значение 42 в поле answer класса Data
4. Запишет какую-то строку в поле question класса Data
5. Запишет значение 9000 в поле maxAllowedValue класса Data
6. Запишет только что созданный объект в поле data класса Keeper

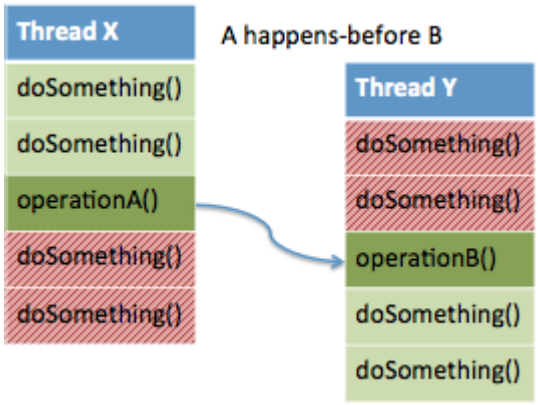
Ничто не мешает другому потоку увидеть произошедшее в пункте 6 до того, как он увидит произошедшее в пунктах 3-5. В результате этот поток увидит объект в некорректном состоянии, когда его поля ещё не были установлены. Такое, разумеется, никого не устроит, и потому есть жёсткий набор правил, по которым оптимизатору/компилятору запрещено выполнять reordering.

Happens-before

Все эти правила заданы с помощью так называемого отношения happens-before. Определяется оно так:

Пусть есть поток X и поток Y (не обязательно отличающийся от потока X). И пусть есть операции A (выполняющаяся в потоке X) и B (выполняющаяся в потоке Y).

В таком случае, A happens-before B означает, что все изменения, выполненные потоком X до момента операции A и изменения, которые повлекла эта операция, видны потоку Y в момент выполнения операции B и после выполнения этой операции.
Начнём с самого простого случая, когда поток только один, то есть X и Y — одно и то же. Внутри одного потока, как мы уже говорили, никаких проблем нет, потому операции имеют по отношению к друг другу happens-before в соответствии с тем порядком, в котором они указаны в исходном коде (program order).



Здесь слева зелёным помечены те операции, которые гарантированно увидит поток Y, а красным — те, что может и не увидеть. Справа красным помечены те операции, при исполнении которых ещё могут быть не видны результаты выполнения зелёных операций слева, а зелёным — те, при исполнении которых уже всё будет видно. Важно заметить, что отношение happens-before транзитивно, то есть если A happens-before B и B happens-before C, то A happens-before C.

Операции, связанные отношением happens-before

Посмотрим теперь, что же именно за ограничения на reordering есть в JMM. Глубокое и подробное описание можно найти, например, в The JSR-133 Cookbook. Начнём с самого простого и известного: блокировок.

1. Освобождение (releasing) монитора happens-before заполучение (acquiring) того же самого монитора. Обратите внимание: именно освобождение, а не выход, то есть за безопасность при использовании wait можно не беспокоиться.
Как исправить данный пример? В данном случае всё очень просто: достаточно убрать внешнюю проверку и оставить синхронизацию как есть. Теперь второй поток гарантированно увидит все изменения, потому что он получит монитор только после того, как другой поток его отпустит. А так как он его не отпустит, пока всё не проинициализирует, мы увидем все изменения сразу, а не по отдельности:

```
public class Keeper {
   private Data data = null;
   public Data getData() {
      synchronized(this) {
         if(data == null) {
            data = new Data();
         }
      }
      return data;
   }
}
```

2. Запись в volatile переменную happens-before чтение из той же самой переменной. Изменение исправляет некорректность, но возвращает того, кто написал изначальный код, туда, откуда он пришёл — к блокировке каждый раз. Спасти может ключевое слово

volatile. Фактически, рассматриваемое утверждение (2) значит, что при чтении всего, что объявлено volatile, мы всегда будем получать актуальное значение. Кроме того для volatile полей запись всегда (в т.ч. long и double) является атомарной операцией. Ещё один важный момент: если у вас есть volatile сущность, имеющая ссылки на другие сущности (например, массив, List или какой-нибудь ещё класс), то всегда «свежей» будет только ссылка на саму сущность, но не на всё, в неё входящее.
С использованием volatile исправить ситуацию можно так:

```
public class Keeper {
   private volatile Data data = null;
   public Data getData() {
      if(data == null) {
         synchronized(this) {
            if(data == null) {
               data = new Data();
            }
         }
      }
      return data;
   }
}
```

Здесь по-прежнему есть блокировка, но только в случае, если data == null. Остальные случаи мы отсеиваем, используя volatile read. Корректность обеспечивается тем, что volatile store happens-beforevolatile read, и все операции, которые происходят в конструкторе, видны тому, кто читает значение поля.

3. Запись значения в final-поле (и, если это поле — ссылка, то ещё и всех переменных, достижимых из этого поля (dereference-chain)) при конструировании объекта happens-before запись этого объекта в какую-либо переменную, происходящая вне этого конструктора.
Это тоже выглядит довольно запутанно, но на самом деле суть проста: если есть объект, у которого есть final-поле, то этот объект можно будет использовать только после установки этого final-поля (и всего, на что это поле может ссылаться). Не стоит, впрочем, забывать, что если вы передадите из конструктора ссылку на конструируемый объект (т.е. this) наружу, то кто-то может увидеть ваш объект в недостроенном состоянии.
В примере достаточно сделать поле, запись в которое происходит последней, final, как всё магически заработает и без volatile и без синхронизации каждый раз:

```
public class Data {
   String question;
   int answer;
   final int maxAllowedValue;
   public Data() {
      this.answer = 42;
      this.question = reverseEngineer(this.answer);
      this.maxAllowedValue = 9000;
   }
   private String reverseEngineer(int answer) {
      return null;
   }
}
```

Кроме того, важно помнить, что поля бывают ещё и статические, а что инициализацию классов JVM гарантированно выполняет лишь один раз при первом обращении.

```
public class Singleton {
   private Singleton() {}
   private static class InstanceContainer {
      private static final Singleton instance = new Singleton();
   }
   public Singleton getInstance() {
      return InstanceContainer.instance;
   }
}
```

Это не является советом к тому, как нужно реализовывать синглтон.

Материалы для подготовки

1. https://www.youtube.com/watch?v=hxIRyqHRnjE
2. https://shipilev.net/#jmm
3. Обзор собеседования Concurrency http://www.duct-tape-architect.ru/?p=294 https://habrahabr.ru/post/164487/
4. https://habrahabr.ru/company/luxoft/blog/157273/

Вопросы для самоконтроля

1) Перечислите основные способы запуска кода в потоке
2) Перечислите основные способы синхронизации
3) Что такое Deadlock?
4) Перечисли состав пакета java.util.concurrent
5) В чем назначение модели памяти в Java?